4. Recursion

- Motivation
- Problem Solving Tips
- Factorial
- Array Reversal
- Sum Squares
- Calculates X^p
- Fibonacci Numbers
- Ackerman's Function
- Binary Search
- Linked List Traversal
- Towers of Hanoi
- Recursive Flood Fill
- Defining Languages
  - Languages
  - Palindrome
  - Expressions
- 8 Queens

Motivation

- Recursion is a powerful problem solving tool that is based on mathematical induction
- The idea is to express solution to problem X in term of smaller versions of X
- All recursive solutions can be implemented iteratively, but they often require less code when implemented recursively
- Recursion also gives an easy way to backtrack if a searching algorithm reaches a dead end

Problem Solving Tip

- Think like a manager!
- Take large problem and break into parts to delegate to employees
- Tell employees to think like managers and subdivide their tasks
- Always need a termination condition so employees know when to stop dividing and delegating
- For speed, try to divide the problem in half (or even smaller).

Factorial Example

$$N! = N.\underbrace{(N-1).(N-2)\ldots.3.2.1}_{\text{Really } (N-1)!}$$

$$N! = N.(N-1)!$$

- Hence, solution to N factorial can be written as a smaller factorial problem
- We need a terminating condition to stop this sequence of recursive replacements.

$$\left.\begin{array}{l} 1! = 1 \\ 0! = 1 \end{array}\right\} \text{Common stopping conditions}$$

Factorial Implementation

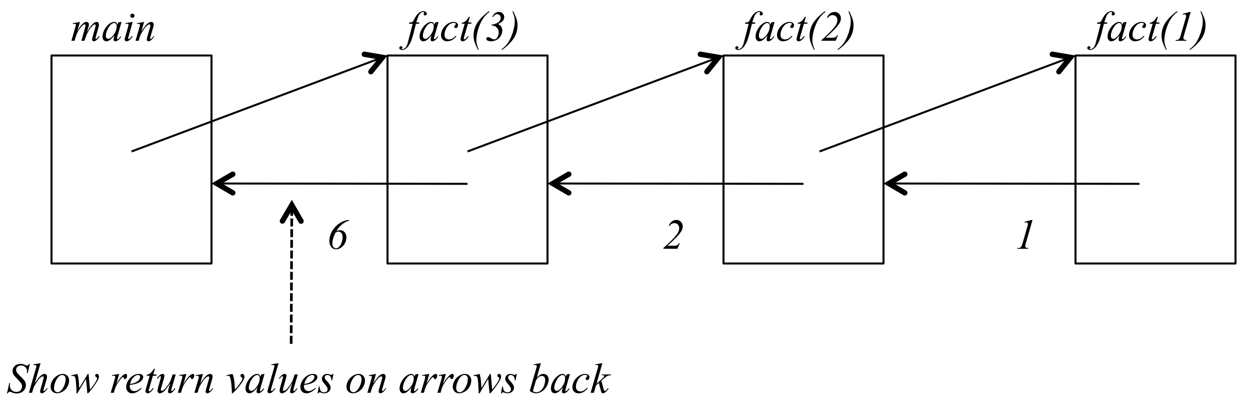
```
int  factorial (int num)
{
  // check terminating condition
  if (num <= 1)
    return 1;
  // handle recursive case
  else
    return (num * factorial(num – 1));
}
```

a smaller problem

- The code above has same number of multiplies as an iterative solution.
- Recursive answer slightly slower due to function call overhead.

Tracing Factorial Execution

- Box method tracing is helpful for showing what recursive function do
- Pretend we have multiple copies of code and draw a box for each

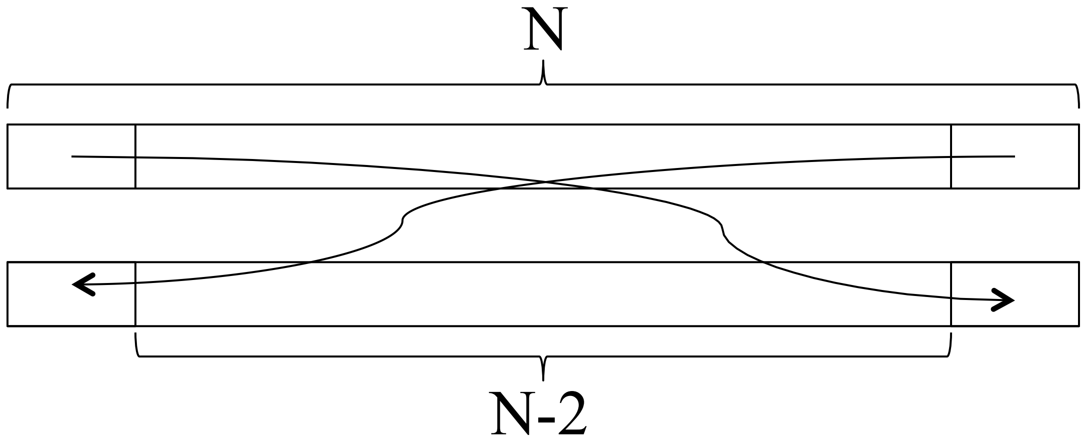


Tracing Factorial Execution

- Draw arrows pointing to new box to show when a function calls itself
- Draw arrows back to the calling function to show return value
- Show function parameters and their values at the top of box
- Show important local variables and their values inside each box

Array Reversal Example

- Assume we are given array of N values to reverse



Algorithm:
- swap first and last elements in array
- recursively reverse N-2 elements in between

- Problem gets smaller at each step
- Stop when asked to swap 0 or 1 elements

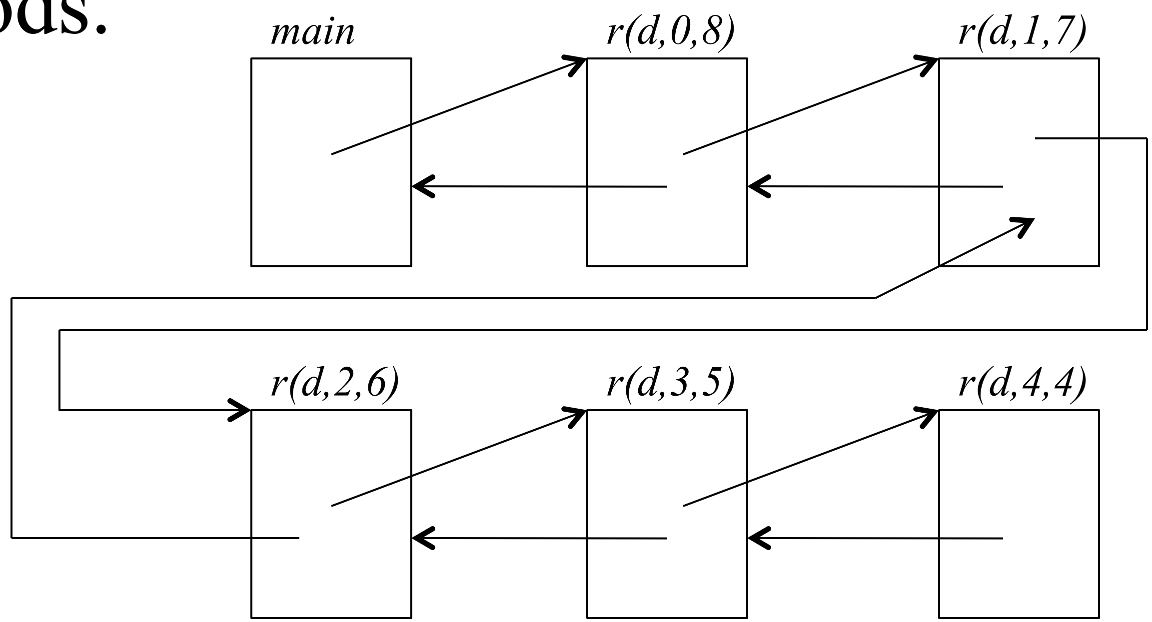
Array Reversal Implementation

```
void reverse(int []data, int low, int high)
{
    // check termination condition
    int size = high – low + 1;
    if (size <= 1)
            return;

    // handle recursive case
    else
    {
            int temp = data[low];
            data[low] = data[high];
            data[high] = temp;

            reverse (data, low + 1, high – 1);
    }
}
```

```
int main ()
{
    int d[9] = {3, 1, 4, 1, 5, 9, 2, 6, 5}
    reverse (d, 0, 8);
    …
}
```

Tracing Array Reversal

- Tracing execution of reverse(d,0,8) using the box methods.



- Each recursive call the array to reverse is two shorter
- What happens if we reverse an odd number of elements?

Sum Squares Example

- Task is to compute sum of squares for given range of integers [low..high]

$$SS = \sum_{i=low}^{high} i^2$$

- Could use iteration
- Could also use recursion and a <u>divide and conquer</u> approach

Sum Squares Example

$$\underbrace{1^2+2^2+3^2+4^2+5^2}_{SS(1,5)}+\underbrace{6^2+7^2+8^2+9^2+10^2}_{SS(6,10)}$$

$$\underbrace{1^2+2^2+3^2}_{SS(1,3)}+\underbrace{4^2+5^2}_{SS(4,5)}$$

$$\underbrace{4^2}_{SS(4,4)}+\underbrace{5^2}_{SS(5,5)}$$

- Each time we divide the range in half and use SS to calculate each half

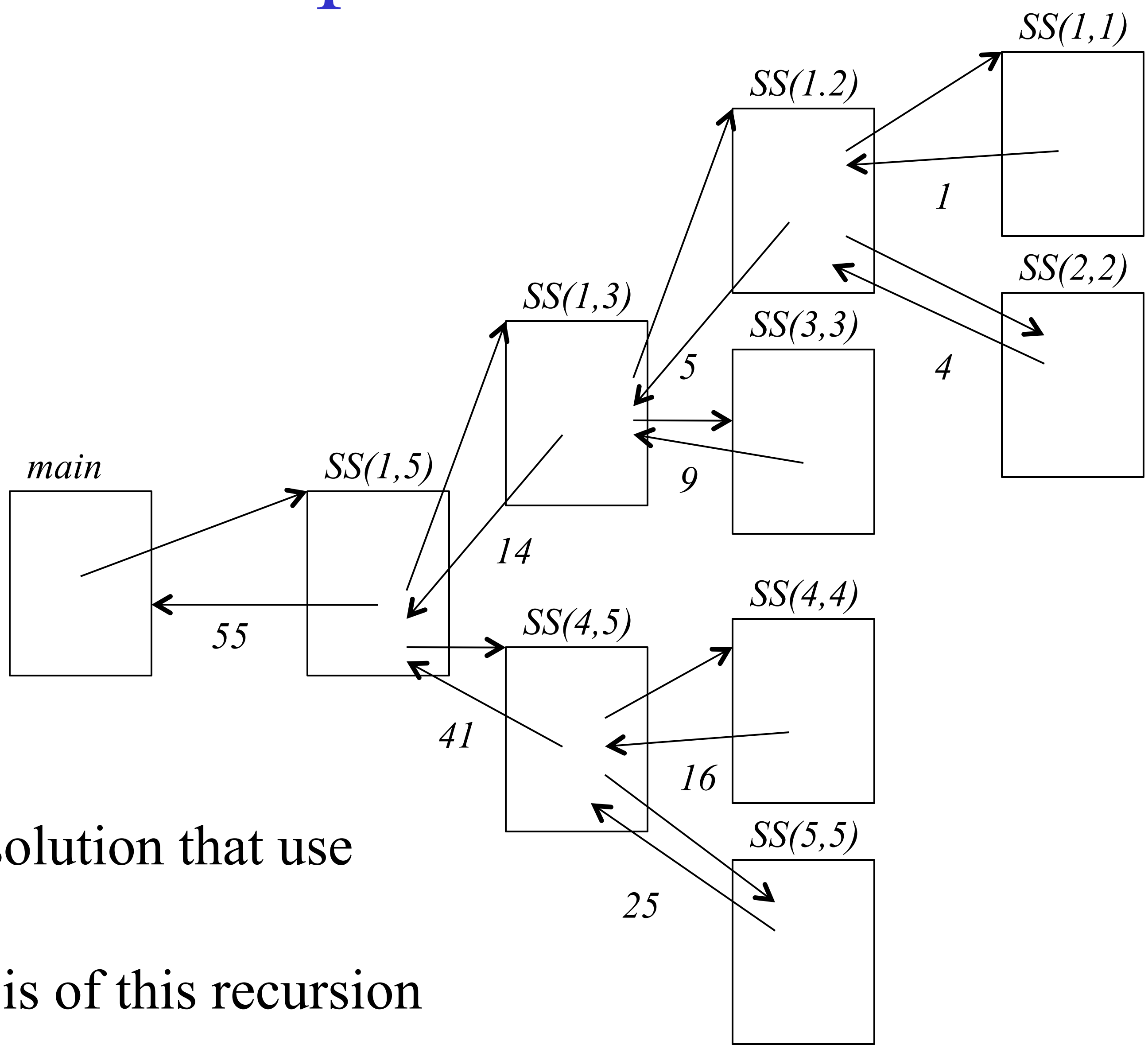
Sum Squares Implementation

```
int sum_squares(int low, int high)
{
    // check termination condition
    if (low == high)
            return low*low;

    // handle recursive case
    else
    {
            int mid = (low + high)/2;
            return (sum_squares(low, mid) +
                 sum_squares(mid, high));
    }
}
```

- Notice that recursion stops when we only have one number to sum
- Notice that each recursive call cuts the size of original program in half (much better than subtracting 1).

Tracing Sum Squares Execution

- Tracing execution of SS(1,5) using the box methods.



- Notice that at each level of recursion the number of boxes doubles

- This is typical for solution that use divide and conquer

- More on the analysis of this recursion later …

Calculation X^p Recursively

- Assume we are given X and p and must compute X^p
- Could do this iteratively by multiplying X by itself p-1 times (p is integer)
- Could also solve recursively using the following:

$$X^p = X^{\frac{p}{2}}.X^{\frac{p}{2}}$$ when p is even

$$X^p = X.X^{\frac{p}{2}}.X^{\frac{p}{2}}$$ when p is odd

(assume p/2 truncates downward)

$$\left.\begin{aligned} X^1 &= X \\ X^0 &= 1 \end{aligned}\right\}$$ terminating conditions

X^p Implementation

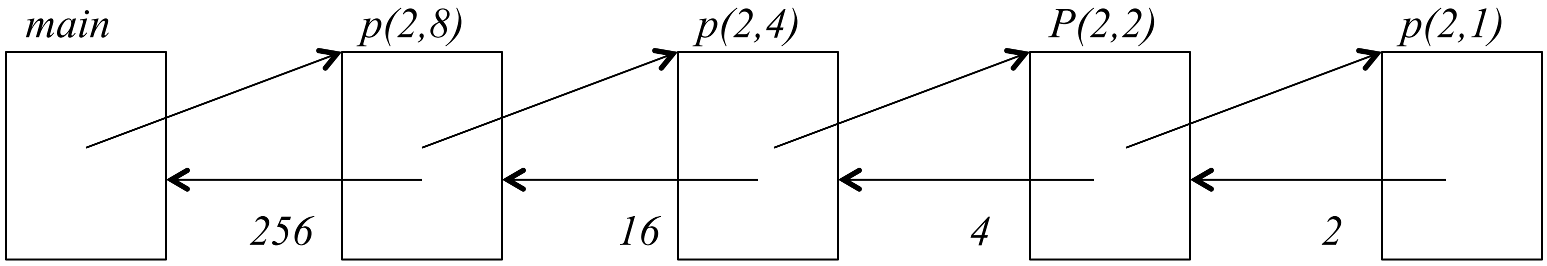
```
float power(float x, int p)
{
    // check termination condition
    if (p == 0)
            return 1;
    else if (p==1)
            return X;

    // handle recursive case
    else if (p%2 == 0)
    {
            float temp = power (X, p/2);
            return temp * temp;
    }
    else if (p%2 == 1)
    {
            float temp = power (X, p/2);
            return X * temp * temp;
    }
}
```

or we could call power(X, 1+p/2)

- Does this code work for all p?

Tracing X^p

- Trace execution of power(2,8)



- Calculated solution with 3 multiplies in stead of 7
- In general, we get answer after $\log_2 p$ steps, which is much better than simple iterative solution

Fibonacci Numbers

- Sequence of numbers that model an explosive growth rate (like rabbit reproduction)

F(N)	1	1	2	3	5	8	13	21	34	55
N	1	2	3	4	5	6	7	8	9	10

- Notice that F(N) = F(N-1) + F(N-2) except at start where F(1) = F(2) = 1
- Possible to write an iterative program to compute F(N) (actually my first program)
- Can also implement recursively

Fibonacci Implementation

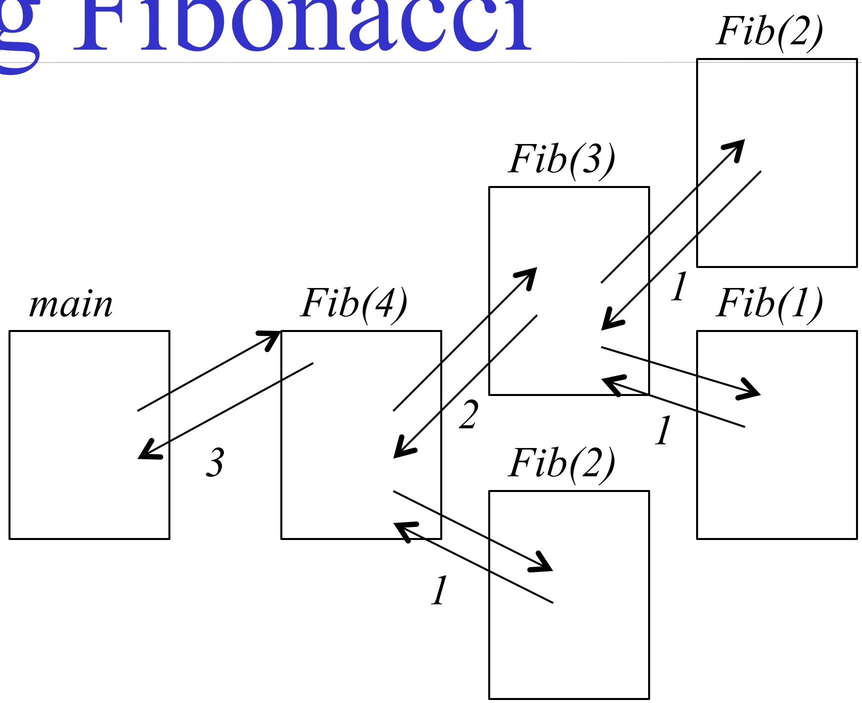
```
int Fib(int num)
{
    // check termination condition
    if (num <= 2)
            return 1;

    // handle recursive case
    else
            return (Fib(num – 1) + Fib(num – 2);
}
```

- Notice two recursive calls in Fib.

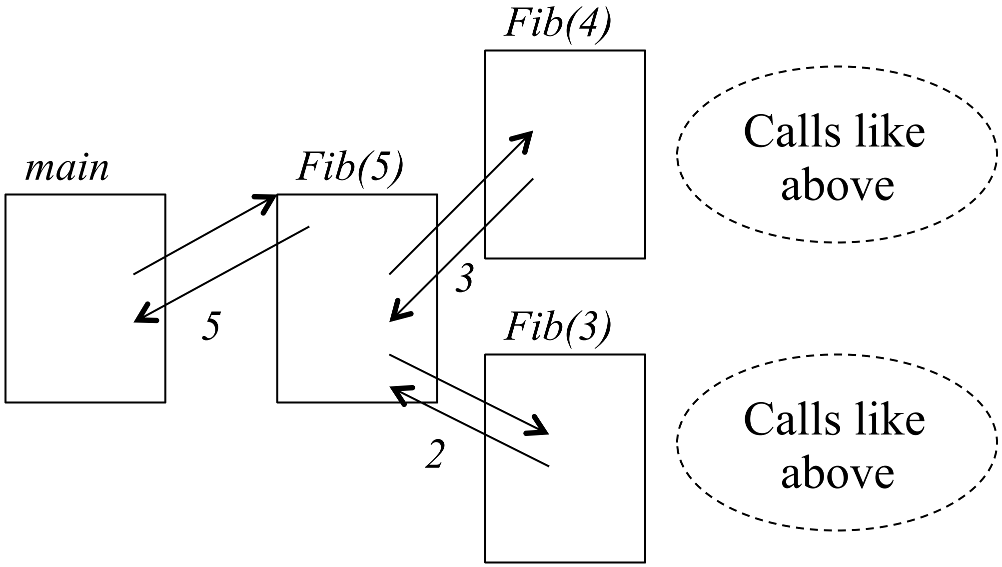
(This will cause an explosion in function calls)

Tracing Fibonacci

- What happens if we call Fib(4)



- What happens if we call Fib(5)



Fibonacci Analysis

- How many recursive function calls are needed to compute F(N)?
- Based on our trace of execution

  calls(N) = calls(N-1) + calls(N-2) + 1

  calls(2) = calls(1) = 1

calls(N)	1	1	3	5	9	15	25	41	67
N	1	2	3	4	5	6	7	8	9

- This is slightly worse than the actual Fibonacci sequence itself

Ackerman's Function

- Function designed to be very recursive and grow rapidly

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } n = 0 \\ A(m-1, A(m, n-1)) & \text{otherwise} \end{cases}$$

Examples:

$A(0,3) = 4$

$A(1,0) = A(0,1) = 2$

$A(1,1) = A(0, A(1,0))$

$\qquad = A(0, 2) = 3$

$A(2,0) = A(1, 1) = 3$

$A(2,1) = A(1, A(2,0)) = A(1,3)$

$\qquad = A(0, A(1,2)) = A(0,4) = 5$

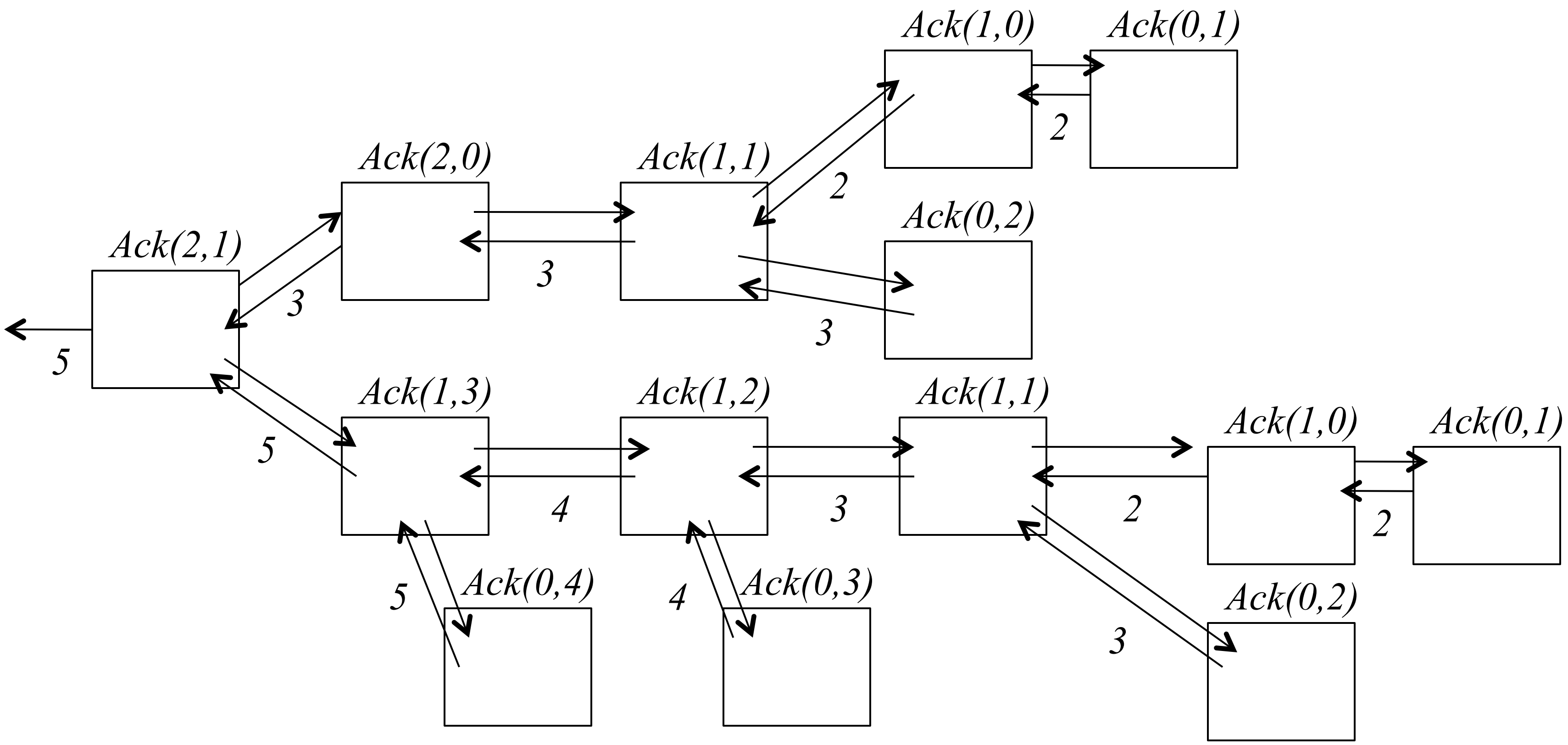
Ackerman Implementation

```
Int Ack(int m, int n)
{
    // check termination condition
    if (m == 0)
        return n+1;

    // handle simple recursion
    else if (n == 0)
        return (Ack(m-1, 1));

    // handle messy recursion
    else
    {
        int temp = Ack(m, n-1);
        return (Ack(m-1, temp));
    }
}
```

- Notice that the m value decreases in recursive calls, but n often increases

Tracing Ackerman

- Consider execution of Ack(2,1)



- Several values are recalculated during this process
- Stack overflow will occur even with small values of m, n

Tracing Ackerman

Entering ackerman 2 1
Entering ackerman 2 0
Entering ackerman 1 1
Entering ackerman 1 0
Entering ackerman 0 1
Leaving ackerman 0 1
Leaving ackerman 1 0
Entering ackerman 0 2
Leaving ackerman 0 2
Leaving ackerman 1 1
Leaving ackerman 2 0
Entering ackerman 1 3
Entering ackerman 1 2
Entering ackerman 1 1
Entering ackerman 1 0
Entering ackerman 0 1
Leaving ackerman 0 1
Leaving ackerman 1 0
Entering ackerman 0 2
Leaving ackerman 0 2
Leaving ackerman 1 1
Entering ackerman 0 3
Leaving ackerman 0 3
Leaving ackerman 1 2
Entering ackerman 0 4
Leaving ackerman 0 4
Leaving ackerman 1 3
Leaving ackerman 2 1

Binary Search

- Assume we are given <u>sorted</u> array of data values.
- What is the fastest way to search for a given value?
- Brute force search scans from L->R
- Better approach is to divide and conquer
- Algorithm:
  - Look at value in middle of the array
  - If less than desired value, search half to right
  - If greater than desired value, search half to left
- Problem is <u>half</u> as large in each recursive step, so algorithm is very fast ($\log_2 N$ steps to search N values)

Binary Search Implementation

```
int search(int []data, int value, int low, int high)
{
    int mid = (low + high)/2;

    // check termination condition
    if (low > high)
        return -1; //not found
    else if (data[mid] == value)
        return mid; //found

    // handle recursive case
    else if (data[mid] > value)
        return search(data, value, low, mid – 1);

    else if (data[mid] < value)
        return search(data, value, mid + 1, high);
}
```

- Notice that we use mid – 1 and mid + 1 in recursive calls to avoid looking at mid again.

Tracing Binary Search

- Search for value 7 in array below

1	3	4	4	5	6	6	7	9	14	16
0	1	2	3	4	5	6	7	8	9	10

search(data, 7, 0, 10)

  mid = (0 + 10)/2 = 5, data[5]<7 so search right

search(data, 7, 6, 10)

  mid = (6 + 10)/2 = 8, data[8]>7 so search left

search(data, 7, 6, 7)

  mid = (6 + 7)/2 = 6, data[6]<7 so search right

search(data, 7, 7, 7)

  mid = (7 + 7)/2 = 7, data[7]=7 so value is found!

Tracing Binary Search

- Search for value 2 in array below

1	3	4	4	5	6	6	7	9	14	16
0	1	2	3	4	5	6	7	8	9	10

search(data, 2, 0, 10)

mid = (0 + 10)/2 = 5, data[5]>2 so search left

search(data, 2, 0, 4)

mid = (0 + 4)/2 = 2, data[2]>2 so search left

search(data, 2, 0, 1)

mid = (0 + 1)/2 = 0, data[0]<2 so search right

search(data, 2, 1, 1)

mid = (1 + 1)/2 = 1, data[1]>2 so search left

search(data, 2, 1, 0)

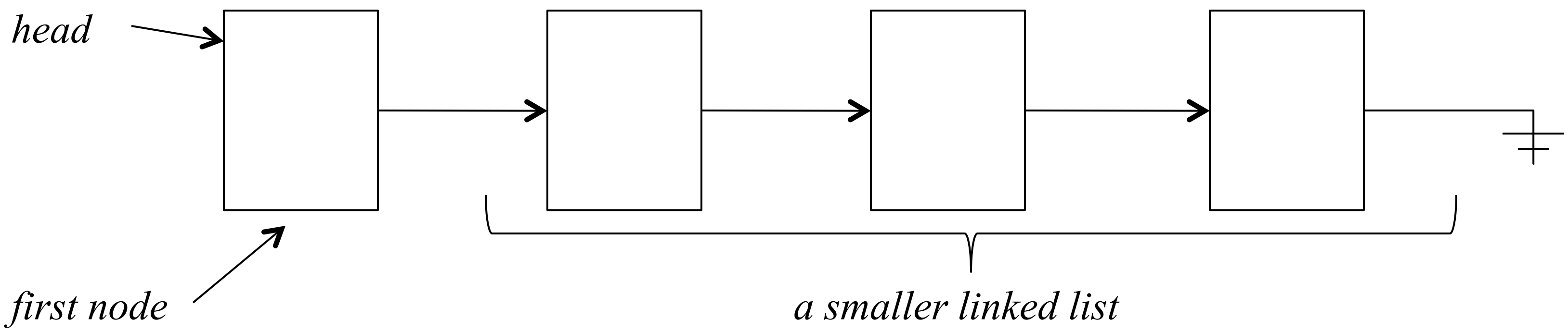
low>high, so data is not found!

Tracing Binary Search

- We have tested binary search looking for a value that does exist in the sorted array
- We have also tested case where value is not found in the sorted array
- Several other cases to consider:
  - Search for values in data[0] and data[N-1]
  - Search for value <u>less than</u> data[0]
  - Search for value <u>greater than</u> data[N-1]

Linked List Traversal

- A linked list is sometimes called a <u>recursive data type</u>



- Hence to traverse a list, we can visit first node and then call the traverse function recursively to process nodes after first node.
- Need to terminate process when we have an empty list.

Recursive List Print

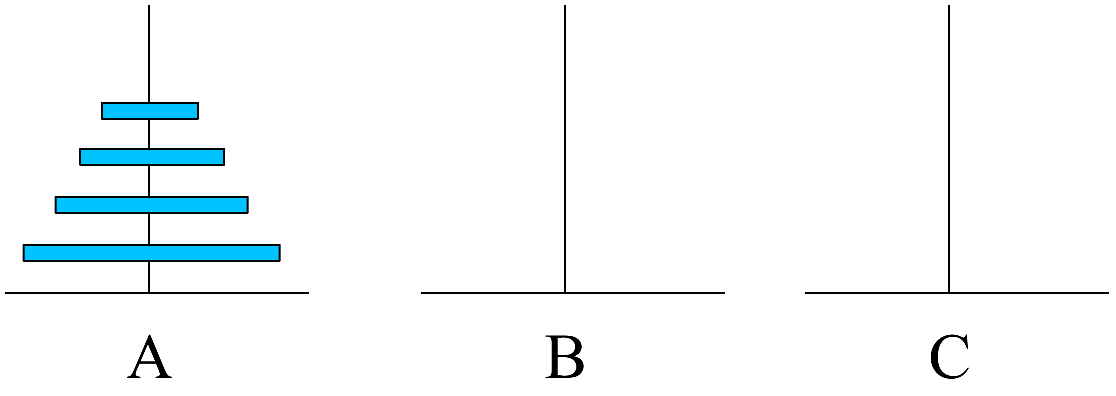
- Assume we have data node declared as a struct with “value” and “next” fields

```
void print(Node *prt)
{
    // handle terminating condition
    if (ptr == NULL)
        return;

    // print value
    cout << “value=“ << prt->value << endl;

    // handle recursion
    print(ptr->next);
}
```

- This function will make one recursive call for each node in the linked list. (Notice there is no while loop above!).
- What happens if we reverse the cout and recursive print call above??

Towers of Hanoi

- A classic puzzle
- Start with three pegs and a stack of disks on first peg.
- Disks are of increasing sizes with the smallest on top and largest on bottom.
- Goal is to move 1 disk at a time from one peg to another and end up with all disks on last peg.
- Rule: You cannot put a larger disk on top of a smaller disk.



Example

A	3 2 1
B	
C	

A	3 2
B	
C	1

A	3
B	2
C	1

A	3
B	2 1
C	

A	
B	2 1
C	3

A	1
B	2
C	3

A	1
B	
C	3 2

A	
B	
C	3 2 1

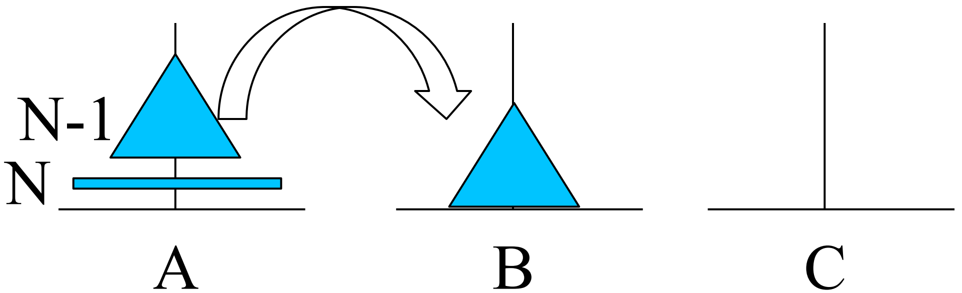
done !

- Notice that at one point we moved the 2, 1 pile out of the way, moved the 3 and then put 2, 1 back
- This is key to a recursive solution because 2, 1 pile is smaller

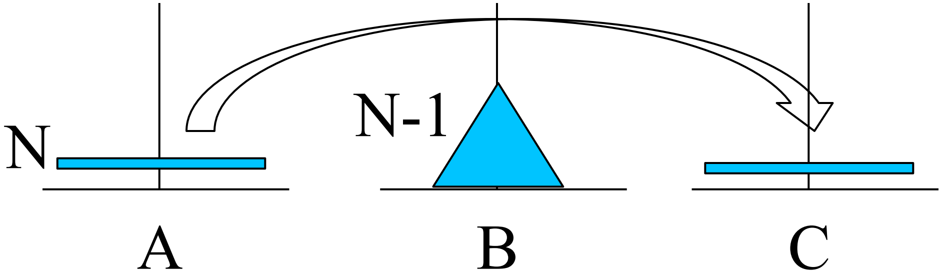
Hanoi Algorithm

- Assume task is to move N disks from peg A onto peg C.

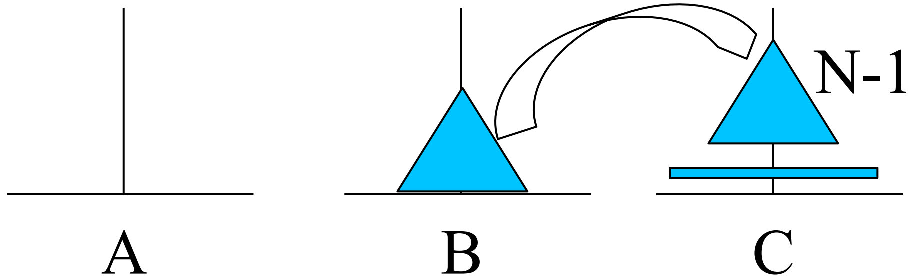
1. Move N – 1 disks from A to B



2. Move Nth disk from A to C



3. Move N-1 disks from B to C



done! (just need to find someone to move the N-1 disks around

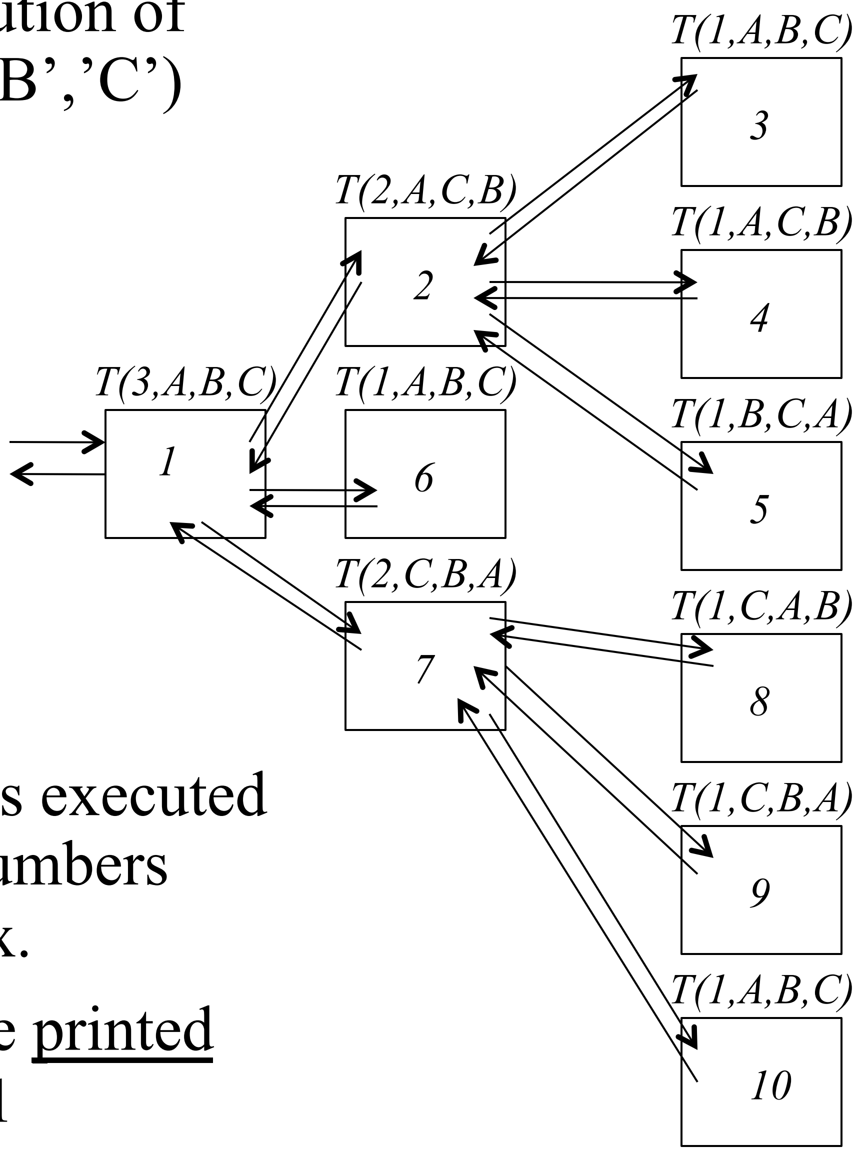
Hanoi Implementation

```
void Tower(int count, char src, char dest, char extra)
{
    // handle terminating condition
    if (count == 1)
        cout << “move disk from” << src << “to” << dest;

    // handle recursive case
    else
    {
        Tower(count – 1, src, extra, dest);
        Tower(1, src, dest, extra);
        Tower(count – 1, extra, dest, src);
    }
}
```

- Code above will move N disks from src to dest using extra as temporary peg.
- Instructions on how to move disks are printed as program executes.

Tracing Hanoi

- Consider execution of Tower(3, ‘A’,’B’,’C’)



- Order functions executed indicated by numbers inside each box.
- Instructions are <u>printed</u> when count = 1

Hanoi Output

move disk from A to B (3)

……………… A to C (5)

……………… A to C (6)

……………… A to C (7)

……………… A to C (8)

……………… A to C (9)

……………… A to C (10)

- First 3 lines move 2 disk from A to C
- Next line moves 1 disk from A to B
- Last 3 lines move 2 disks from C to B

Hanoi Analysis

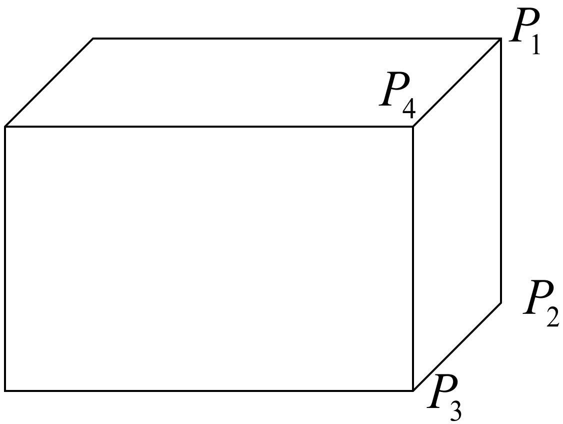
- How many moves are needed to move N disks?

N	Moves(N)
1	1
2	2.Moves(1) + 1 = 3
3	2.Moves(2) + 1 = 7
4	2.Moves(3) + 1 = 15
5	2.Moves(4) + 1 = 31

- In general, Moves(N) = $2^N - 1$
- Thus if N = 20, over a million moves are needed to move disks!
- This is a classic exponential algorithm

Recursive Flood Fill

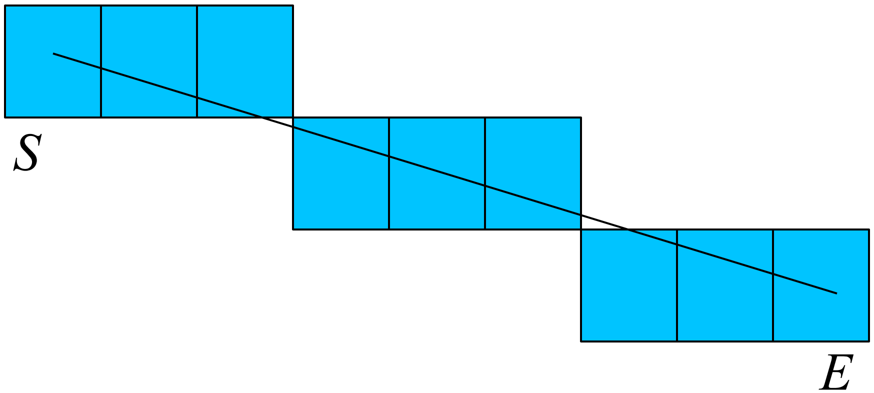
- In computer graphic objects are often modeled by a collection of polygons (surfaces defined by a sequence of endpoints)

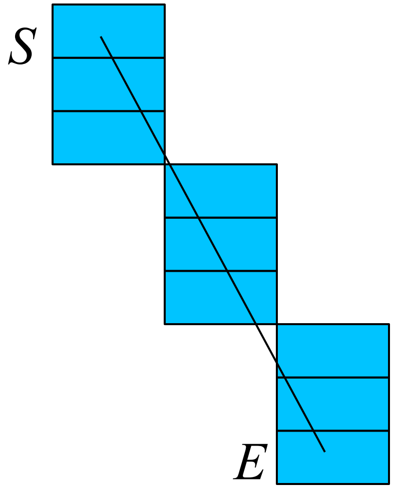


- To draw the object, programs are written to project 3D points to 2D screen coordinates, and the region inside each polygon is filled with color
- The recursive flood fill algorithm is one way to color the polygon once the polygon boundary is drawn
- We need a seed point inside the polygon to start algorithm. It stops when all pixels inside have been colored

Drawline Algorithm

- If we are given (X_S, Y_S) and (X_E, Y_E), the start and end points of a line, how can we fill the pixels in between?





- The task is to mask all pixels the line intersects between S and E
- When Δx > Δy it is better to loop over x and calculate y coordinates of intersection
- When Δy > Δx it is better to loop over y and calculate x coordinates of intersection
- In both cases, we round to the nearest integer and plot the points

Drawline code

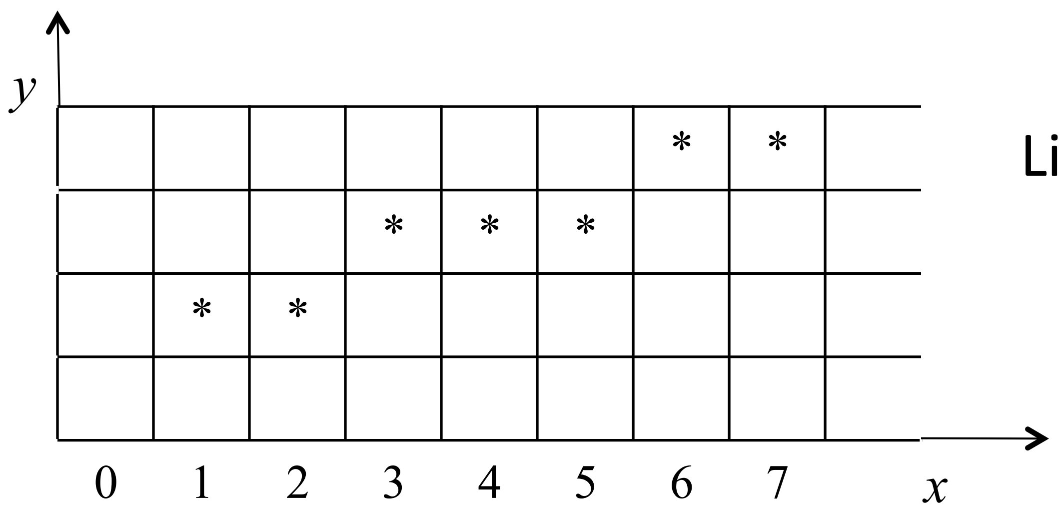
```
void drawline(int color, int Xs, int Ys, int Xe, int Ye)
{
	// calculate slope
	int dX = Xe – Xs;
	int dY = Ye – Ys;
	float slope = (float)dY / (float)dX;

	// handle Δx > Δy case
	if (dX > dY)
	{
		for (int x = Xs; x <=Xe; x++)
		{
			y = (int)(0.5 + (x – Xs) * slope + Ys);
			pixel[y][x] = color;
		}
	}
	else // handle Δy > Δx case
		for (int y = Ys; y <=Ye; y++)
		{
			x = (int)(0.5 + (y – Ys) * slope + Xs);
			pixel[y][x] = color;
		}
	}
}
```

Testing Drawline

- Assume (X_S, Y_S) = (1, 1) and (X_E, Y_E) = (7, 3)
- dX = 6, dY = 2, slope = 0.333

x	y
1	1
2	1
3	2
4	2
5	2
6	3
7	3

- Using the code where dX > dY



Line is filled in with no gaps

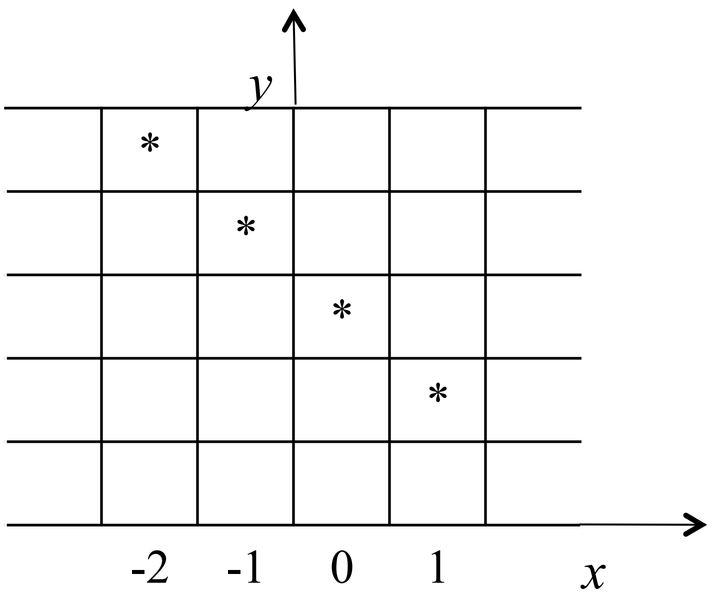
- We assume that line and points are within the array bounds of pixel array.

Testing Drawline

- Assume (X_S, Y_S) = (1, 1) and (X_E, Y_E) = (-2,4)
- dX = -3, dY = 3, slope = -1

x	y
1	1
2	0
3	-1
4	-2

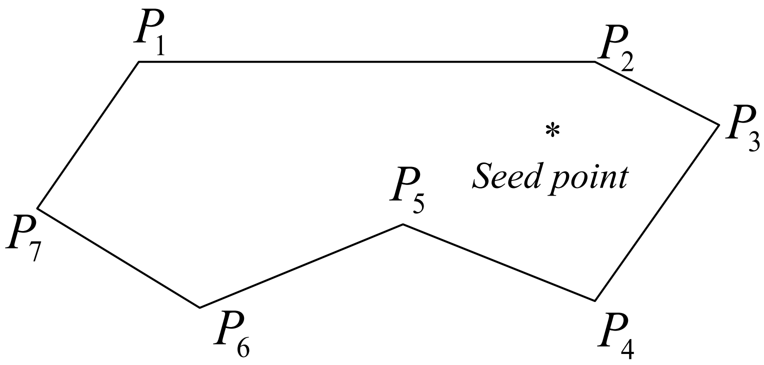
- Using the code where dY >= dX



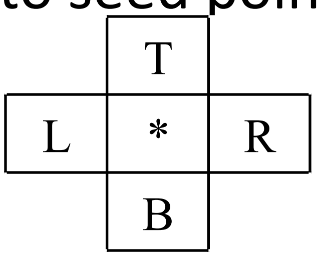
Line is filled in with no gaps

- We assume that negative array indices are handled somehow by magic ….

Flood Fill Algorithm



- Assume we have drawn the boundary of polygon by drawing lines P1->P2, P2->P3, etc…
- Assume we are given (x,y) coordinates of seed point inside the polygon.
- We can grow the region by adding points that are connected to seed point.



- These four neighbors can then be treated as seed points and we can grow the region recursively

- We must take care to stop on boundary or a previously marked pixel in the region.

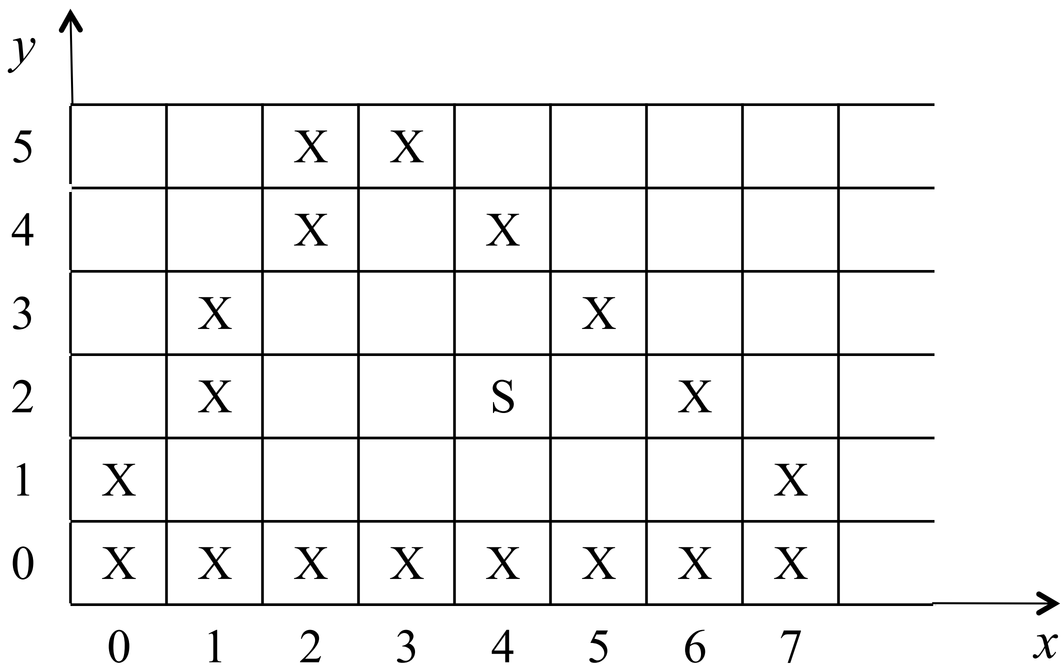
Flood Fill Code

```
void fill(int color, int x, int y)
{
    // handle terminating condition
    if (pixel[y][x] == color)
        return;

    // handle recursive case
    else
    {
        pixel[y][x] = color;
        fill(pixel, color, x + 1, y);    // R
        fill(pixel, color, x – 1, y);    // L
        fill(pixel, color, x , y + 1);   // T
        fill(pixel, color, x , y - 1);   // B
    }
}
```

- This code does no array bounds checking and will die unless boundary is properly drawn.

Testing Flood Fill

- Assume we have 2D array with polygon boundary drawn as show.



- Start by calling fill(4,2)
- Will recursively call fill(5,2)
- Where will it go next?

Testing Flood Fill

- Final result after recursive flood fill
- Polygon filled in with zigzag pattern.

y									
5			X	X					
4			X	13	X				
3		X	12	10	11	X			
2		X	8	9	S	1	X		
1	X	7	6	5	4	2	3	X	
0	X	X	X	X	X	X	X	X	
	0	1	2	3	4	5	6	7	x

- May cause stack overflow if large polygons are rendered this way.

Defining Languages

- We can define languages by specifying the vocabulary and a grammar that states the rules for the language.
- For programming languages we design the grammar to be simple (so we can write compilers and interpreters).
- A terminal is a single word in the language
- A non-terminal is a symbol that stands for zero or more terminals (e.g: verb_phrase)
- A production rule tell us how we can replace a non-terminal with zero or more terminals/non-terminals
- The start symbol is a non-terminal that is used to start the derivation of all sentences in the language.

Language Example

- We can formally specify the “language” of all valid C++ identifiers as follows.

  <id> = <letter>|<id><letter>|<id><digit>

  <letter> = a|b|…|z|A|B|..|Z|

  <digit> = 0|1|…|9

- The non-terminals are <id>, <letter>, <digit>, and individual characters are terminals
- The RHS of production rules show the sets of terminals/non-terminals that can be used to replace terminal on LHS (we use 1 to separate choices to save space).

  <id> → <id><letter>

  → <id><letter><letter>

  → <letter><letter><letter>

  → foo

- Sample derivation of variable “foo”.

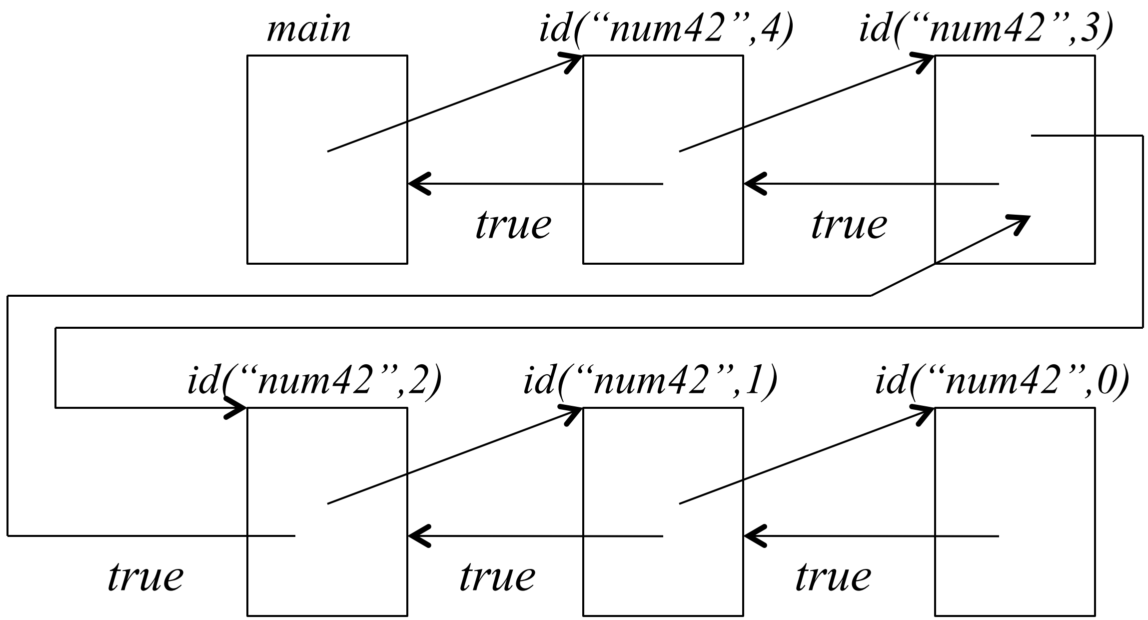
Identifier Code

```
bool id(char []str, int pos)
{
	// handle single char case
	if ((pos == 0) && letter(str[pos]))
		return true;

	// handle recursive case
	else if (letter(str[pos]) || digit(str[pos]))
		return ( id(str, pos – 1));
	  else
		return false; //illegal character
}
```

- Code above checks last digit in str and makes recursive call only if letter or digit found.
- It terminates if single letter is found or an illegal character is read.
- Would be better to process str from L to R but this takes more elaborate grammar.

Tracing Identifier Code

- Trace execution of id("num42",4)



- Each recursive call processes one character on right hand side of str.
- What happens if pos < 0 in first call?

Palindrome Example

- We can formally specify the “language” of palindrome as follows.

  <pal> = Ɛ|<ch>|a<pal>a|b<pal>b|…|z<pal>z

  <ch> = a|b|…|z

- Because the production rules introduce characters in “pairs” on either end of an existing palindrome, the word will be the same read forwards or backwards

  <pal> → <a><pal><a>

  → ab<pal>ba

  → abƐba

  → abba

- Ɛ is empty string so can be removed

- Sample derivation of “abba”, a musical palindrome from way back …

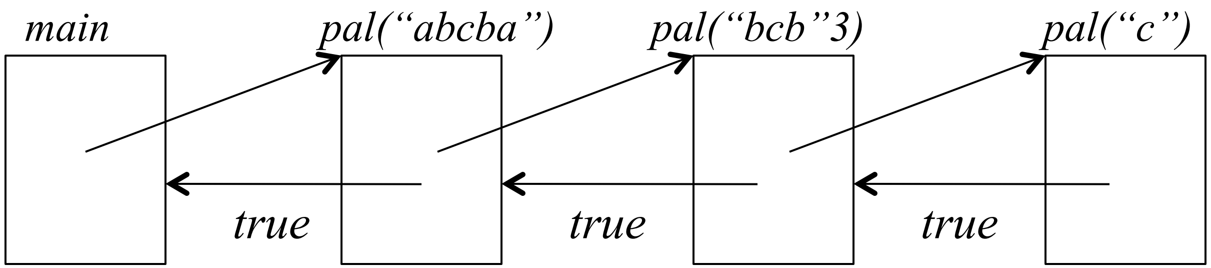
Palindrome Code

```
bool pal(char []str, int low, int high)
{
      // handle single char case
      if (high - low <= 0)
            return true;

      // handle recursive case
      else if (str[low] == str [high])
            return ( pal(str, low + 1, high - 1);
      //otherwise no palindrom
      else
            return false;
}
```

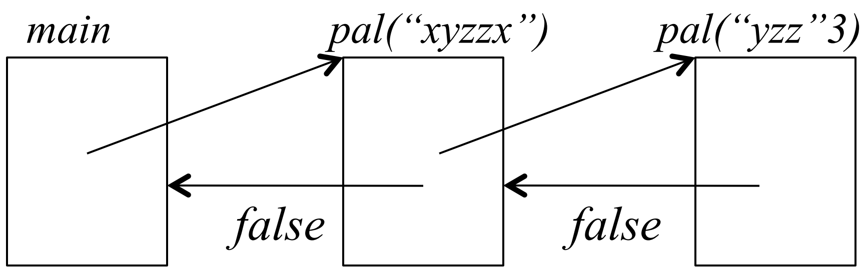
- Code can also be adapted to strings
- Recursive call looks a little like the <pal> production rule.

Tracing Palindrome Code

- Trace execution of pal(“abcba”,0,4)



- Trace execution of pal(“xyzzx”,0,4)



- String is rejected because y ≠ z in substring.

Expression Example

- In order to process expression of the form "3+2*5" we need a grammar that can recognize a sequence of numbers and operators.

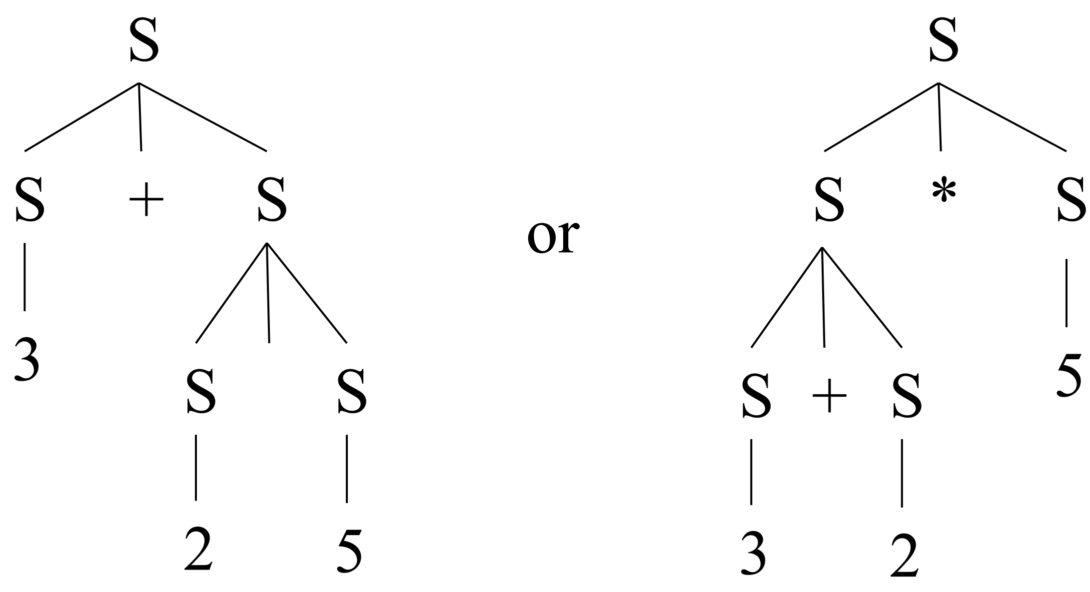
$$S \rightarrow S + S \mid S * S \mid \text{num}$$

$$\begin{aligned} S &\rightarrow S + S \\ &\rightarrow S + S * S \\ &\rightarrow \text{num} + S * S \\ &\rightarrow \text{num} + \text{num} * S \\ &\rightarrow \text{num} + \text{num} * \text{num} \end{aligned}$$

- Derivation for "3 + 2 * 5"

Expression Example

- Can draw derivation in tree form too



- The grammar above is <u>ambiguous</u> because two parse trees are possible (hence two values possible if evaluated this way)

Improved Expression Example

- Can extend grammar to include brackets and impose "multiple before addition" rule

$$S \rightarrow S + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow num \mid (S)$$

$$\begin{aligned} S &\rightarrow S + T \\ &\rightarrow T + T \\ &\rightarrow T + T * F \\ &\rightarrow F + T * F \\ &\rightarrow F + F * F \\ &\rightarrow num + num * num \end{aligned}$$

- Three non-terminal now, S, T, F.
- Only one possible derivation for "3 + 2 * 5" now

- This grammar can be easily extended to include "-" and "÷" operations by adding rules for expanding S and T
- Writing a recursive program to recognize expressions is a little tricky because grammar above is "left recursive".

Recursive Expression Parser

- We can rewrite previous grammar to be "right recursive" as follow

$$
\begin{aligned}
S &\rightarrow TR_1 \\
R_1 &\rightarrow +TR_1 \mid \varepsilon \\
T &\rightarrow FR_2 \\
R_2 &\rightarrow *FR_2 \mid \varepsilon \\
F &\rightarrow num \mid (S)
\end{aligned}
$$

- Here ε stands for empty string.

- Now we can write <u>recursive descent</u> parser with functions called S, R_1, T, R_2, and F as follows.

  S() – call T and R1

  R_1() – check for +, call T and R_1

  T() – call F and R_2

  R_2() – check for *, call F and R_2

  F() – check for digits and bracketed expression

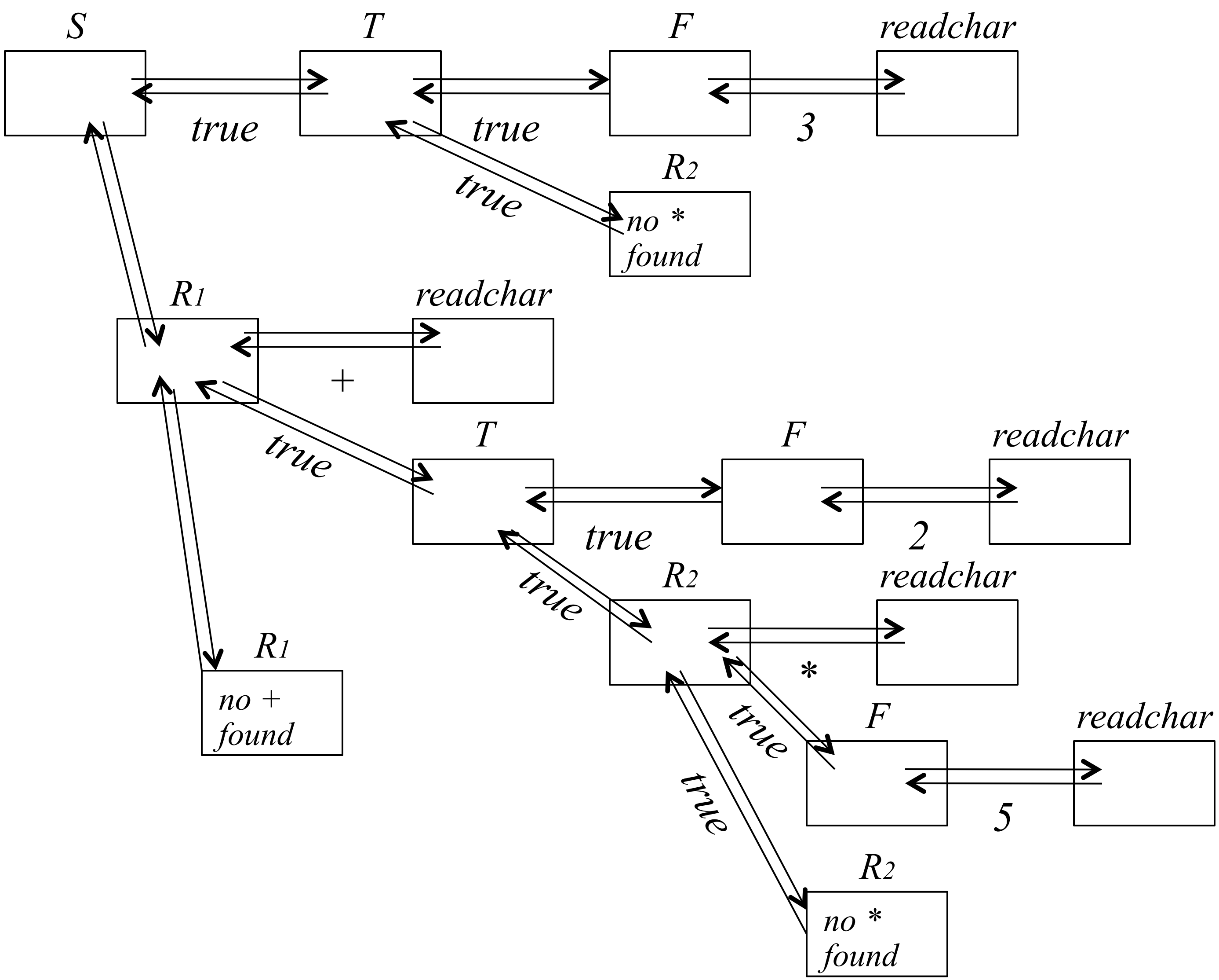
Recursive Parser Implementation

```
bool S()
{
    if ( T() )
        return R1();
    else
        return false;
}
bool R1()
{
    if ( nextchar() == '+')
    {
        readchar();
        if ( T() )
            return R1();
        else
            return false;
    }
    return true;
}
bool F()
{
    if ( (nextchar() >= '0' ) && (nextchar() <= '9'))
    {   readchar(); return true;   }
    else
        return false;
}
```

```
bool T()
{
    if ( F() )
        return R2();
    else
        return false;
}
bool R2()
{
    if ( nextchar() == '*')
    {
        readchar();
        if ( F() )
            return R2();
        else
            return false;
    }
    return true;
}
```

Tracing Recursive Expression Parser

- Trace execution as S() processes “3+2*5”



Languages Discussion

- Clearly the task of parsing and compiling an entire program is beyond the scope of this class.
- Important formal methods for defining and processing grammars is discussed in “formal languages”
- The implementation of programming languages is typically covered in graduate level “compiler writing” classes or capstone projects.

8 Queens Problem

- Given an 8x8 chess board and 8 queens, position the queens 1 per row/column so no queen can take another

*	*	Q	*	*	*	*	*
*	*	*	*	*	Q	*	*
*		*		*	*	*	
		*	*		*		*
		*			*	*	
	*	*			*		*
*		*			*		
		*			*		

- After we place two queens a large number of spaces are no longer “safe”.

- In general, there are 8! possible combinations to consider (40,320)
- Is there a way to shorten this search?
- Yes, using recursion and backtracking …

8 Queen's Algorithm

- Assume that columns 1 to k-1 are solved and queens are placed properly.
- Pick one position in column k that is "safe" from all other queens.
- Recursively try to solve remaining columns with a queen in this position.
- If recursive solution returns success then we are done and can return success.
- Else we need to move column k queen to next "safe" position and try to solve remaining columns again. (this is the backtracking part)
- If no "safe" positions can be found on column k then return failure (and let one of the lower columns backtrack).
- By limiting our search to "safe" positions the search time is much less than 8!

8 Queens Code

```
bool solve(int col)
{
    // check terminating condition
    if (col >= SIZE)
        return true;

    else // handle recursive case
    {
        // try all possible rows
        for (int row = 0; row < SIZE; row ++)
        {
            if (safe(row, col))
            {
                board[row][col] = ‘Q’; //move
                if (solve(col + 1))
                    return true;
                else
                    board[row][col] = ‘   ’; //backtrack
            }
        }
        //return false if no solution found
        return false;
    }
}
```

Testing 8 Queens

Q	*	*	*	*	*	*	*
*	*						
*		*					
*			*				
*				*			
*					*		
*						*	
*							*

col = 0

Q	*	*	*	*	*	*	*
*	*	*					
*	Q	*	*	*	*	*	*
*	*	*	*				
*	*		*	*			
*	*			*	*		
*	*				*	*	
*	*					*	*

col = 1

Q	*	*	*	*	*	*	*
*	*	*			*		
*	Q	*	*	*	*	*	*
*	*	*	*				
*	*	Q	*	*	*	*	*
*	*	*	*	*	*		
*	*	*		*	*	*	
*	*	*			*	*	*

col = 2

Q	*	*	*	*	*	*	*
*	*	*	Q	*	*	*	*
*	Q	*	*	*	*	*	*
*	*	*	*		*		
*	*	Q	*	*	*	*	*
*	*	*	*	*	*		*
*	*	*	*	*	*	*	
*	*	*	*		*	*	*

col = 3

Testing 8 Queens

Q	*	*	*	*	*	*	*
*	*	*	Q	*	*	*	*
*	Q	*	*	*	*	*	*
*	*	*	*	Q	*	*	*
*	*	Q	*	*	*	*	*
*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	*

col = 4

Q	*	*	*	*	*	*	*
*	*	*	Q	*	*	*	*
*	Q	*	*	*	*	*	*
*	*	*	*	*	*		
*	*	Q	*	*	*	*	*
*	*	*	*	*	*	*	*
*	*	*	*	*	*	*	
*	*	*	*	Q	*	*	*

col = 5

No safe rows in col=5, so backtrack on col = 4

Still no safe rows in col = 5, so back track on col = 3 now

Q	*	*	*	*	*	*	*
*	*	*	*		*		
*	Q	*	*	*	*	*	*
*	*	*	*			*	
*	*	Q	*	*	*	*	*
*	*	*	*	*	*		
*	*	*	Q	*	*	*	
*	*	*	*	*	*	*	*

col = 3

Q	*	*	*	*	*	*	*
*	*	*	*	Q	*		
*	Q	*	*	*	*	*	*
*	*	*	*	*		*	
*	*	Q	*	*	*	*	*
*	*	*	*	*	*		
*	*	*	Q	*	*	*	
*	*	*	*	*	*	*	*

col = 4

Now col = 5 has openings so we can continue search